COMS20001 lab. worksheet #6

- Both the hardware and software in MVB-2.11 is managed by the IT Services Zone E team. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: either talk to them directly in room MVB-3.41, or submit a service request online via

  http://servicedesk.bristol.ac.uk
- We intend this worksheet to be attempted, at least partially, in the associated lab. session. Your attendance is important, since this session represents the primary source of formative feedback and help for COMS20001. Perhaps more so than in units from earlier years, *you* need to actively ask questions of and seek help from either the lectures and/or lab. demonstrators present.
- The questions are roughly classified as either L (for coursework related questions that should be completed in the lab. session), or A (for additional questions that are entirely optional). Keep in mind that we only *expect* you to complete the first class of questions: the additional content has been provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring it (since it is not directly assessed).

Before you start work, download (and, if need be, unarchive[a]) the file

http://tinyurl.com/y72dnlcy/csdsp/os/sheet/lab-6_q.tar.gz

somewhere secure[b] in your file system: it is intended to act as a starting point for your own work, and will be referred to in what follows.

[a]Use the `gz` and `tar` commands within a BASH shell (e.g., in a terminal window), *or* the archive manager GUI (available either via the menu `Applications`→`Accessories`→`Archive Manager` or by directly executing `file-roller`) if you prefer.
[b]For example, the `Private` sub-directory in your home directory.

Q1[A]. By this point, we assume you will prefer to focus on the (assessed) coursework assignment rather than the (non-assessed) lab. worksheets, so this question is entirely optional. Put another way, it focuses on a topic related to the final stage of the assignment and is thus more challenging than previous worksheets: unless you aim to submit a solution for the associated option, it makes sense to ignore this question and spend your time working on the rest of the assignment instead.

In the following Sections, the goal is to offer, via another example kernel image, an introduction to the LCD [1] and PS/2 keyboard and mouse [2] controllers and hence a route toward implementation of *graphical* vs. *textual* user interfaces.

Background

The LCD controller Although less capable than a consumer GPU, the PL111[1] LCD controller still supports a fairly broad set of functionality. It remains conceptually simple, however: it acts as an intermediary between the pixel-based display hardware (i.e., LCD screen) and any user of it (i.e., the kernel), offering a programming interface to abstract all physical, electronic details of how the display itself operates. You could think of the controller as operating in a loop. In each iteration, it inspects a region of memory termed the frame buffer; the content is interpreted as virtual pixels it then uses to control associated, physical pixels on the display.

Consider a display whose resolution is $w \times h$, meaning it is w pixels wide (in the x-dimension) and h pixels high (in the y-dimension); concretely, say $w = 800$ and $h = 600$ meaning a total of $800 \cdot 600 = 480000$ pixels. You could formalise the pixels in such a display using a $(h \times w)$-element matrix

$$P = \begin{pmatrix} P_{0,0} & P_{0,1} & \dots & P_{0,w-1} \\ P_{1,0} & P_{1,1} & \dots & P_{1,w-1} \\ \vdots & \vdots & \ddots & \vdots \\ P_{h-1,0} & P_{h-1,1} & \dots & P_{h-1,w-1} \end{pmatrix}$$

where $P_{i,j}$ denotes a pixel in the i-th row and j-th column. The controller is therefore tasked with inspecting the frame buffer content, and using it to control each $P_{i,j}$ st. it displays the correct colour.

[1] QEMU uses an emulated implementation of this device detailed in http://github.com/qemu/qemu/blob/master/hw/display/pl110.c. Keep in mind that the implementation is incomplete; some features of the PL111, for example the hardware cursor, are not supported.

This demands we understand how the frame buffer should be structured, so the controller correctly interprets the content. Although others are supported, consider the 16 BPP and 5 : 5 : 5 formats (which are used for TFT-based displays per [1, Section 1.1.7]). In easier terms, this just means a 16 bit per pixel colour depth, st. each $P_{i,j}$ is controlled using a 16-bit integer, and each of the red (r), green (g), and blue (b) channels are controlled using 5 of those 16 bits st.

$$P_{i,j} = 0 \parallel b \parallel g \parallel r$$

for 5-bit b, g and r: we can see, for example, that

$$\begin{array}{lclllll} 7FFF_{(16)} & = & 0_{(2)} & \parallel 11111_{(2)} & \parallel 11111_{(2)} & \parallel 11111_{(2)} & \mapsto \text{white} \\ 0000_{(16)} & = & 0_{(2)} & \parallel 00000_{(2)} & \parallel 00000_{(2)} & \parallel 00000_{(2)} & \mapsto \text{black} \\ 001F_{(16)} & = & 0_{(2)} & \parallel 00000_{(2)} & \parallel 00000_{(2)} & \parallel 11111_{(2)} & \mapsto \text{red} \\ 03E0_{(16)} & = & 0_{(2)} & \parallel 00000_{(2)} & \parallel 11111_{(2)} & \parallel 00000_{(2)} & \mapsto \text{green} \\ 7C00_{(16)} & = & 0_{(2)} & \parallel 11111_{(2)} & \parallel 00000_{(2)} & \parallel 00000_{(2)} & \mapsto \text{blue} \end{array}$$

So, using this format, the kernel can allocate a 480000-element array of 16-bit integers and configure the controller with the base address (so it knows where to load content from). Either 1-dimensional, i.e.,

```
uint16_t fb[ 480000 ];
```

or 2-dimensional, i.e.,

```
uint16_t fb[ 600 ][ 800 ];
```

organisations[2] are possible, but, either way, the idea is that an element in `fb` will control some corresponding pixel in P. For example, using the 2-dimensional organisation, the assignment

```
fb[ 10 ][ 20 ] = 0x001F;
```

sets pixel $P_{10,20}$ to a (pure) red colour.

The PS/2 controller The PS/2[3] (port *and* protocol) standard was initially developed by IBM as a means of interfacing low-speed I/O devices (more specifically, keyboards and pointing devices, i.e., mice) with computer systems; it superseded serial-based (cf. UART) approaches, and was in turn superseded by modern alternatives such as USB.

Perhaps more so that USB for example, which arguably suffers due to the generality it must support, PS/2 is relatively easy to use. The PL050 PS/2 controller (or Keyboard and Mouse Interface (KMI) in ARM terminology) facilitates usage, offering a very simple programming interface; the PB-A8 has two such controllers, one reserved (i.e., used by) QEMU for a keyboard and the other for a mouse. Basically, the controller abstracts all physical, electronic detail of how each PS/2 *port* operates by presenting a UART-like, bi-directional byte-oriented communication channel with the connected device. The user (i.e., kernel) can then focus on aligning said communication with the PS/2 *protocol*[4] by interpreting bytes received as key-presses or mouse movement.

Conceptually at least, this protocol is fairly easy to understand; on the lab. workstations, it is *also* easy to deal with in an kernel implementation that uses the controller. However, delivery of key-presses can, in particular, introduce complications that stem from different types of keyboard hardware etc. So, keep in mind that the keyboard mapping, i.e., which scancode[5] QEMU receives when you press a key, matters a *lot*: it will likely differ between platforms you execute QEMU on, meaning work you carry out under MacOS, for example, may not work under Linux (and vice versa) without appropriate porting.

Explore the archive content

As Figure 1 illustrates, the content and structure of the archived material provided matches worksheet #4. Note one difference in the `Makefile`-based build system: `QEMU_DISPLAY` is now set st. execution of QEMU renders the LCD display and accepts keyboard and mouse input in a previously suppressed window.

Understand the archive content

`image.ld`: the linker script Figure 2 illustrates the linker script `image.ld`. It controls how `ld` produces the kernel image from object files, which, in turn, stem from compilation of the source code files; the resulting layout in memory is illustrated by Figure 3.

[2] The controller will inspect the frame buffer in row-major order: you could view this as a by-product of older, CRT-based displays which displayed content row-by-row in what are termed scan lines. For a 1-dimensional organisation this means row i of pixels in P is controlled by elements $i \cdot w$ to $(i+1) \cdot w - 1$ of the frame buffer. This indexing is of course automatically catered for using a 2-dimensional organisation.

[3] Note that the name stems from the IBM Personal System/2 computer; we are *not* talking about a PlayStation2 here!

[4] See `http://wiki.osdev.org/PS/2_Keyboard` and `http://wiki.osdev.org/PS/2_Mouse` for example.

[5] `http://en.wikipedia.org/wiki/Scancode`

```
${ARCHIVE}/lab-6_q
         ├── device
         │     ├── *.h
         │     └── *.c
         ├── kernel
         │     ├── int.h
         │     ├── int.s
         │     ├── lolevel.h
         │     ├── lolevel.s
         │     ├── hilevel.h
         │     └── hilevel.c
         ├── user
         ├── image.ld
         └── Makefile
```

Figure 1: *A diagrammatic description of the material in* `lab-6_q.tar.gz`*.*

```
SECTIONS {
  /* assign load address (per  QEMU) */
  .        =     0x70010000;
  /* place text segment(s)           */
  .text : { kernel/lolevel.o(.text) *(.text .rodata) }
  /* place data segment(s)           */
  .data : {                         *(.data        ) }
  /* place bss  segment(s)           */
  .bss  : {                         *(.bss         ) }
  /* align        address (per AAPCS) */
  .        = ALIGN( 8 );
  /* allocate stack for irq mode     */
  .        = . + 0x00001000;
  tos_irq = .;
  /* allocate stack for svc mode     */
  .        = . + 0x00001000;
  tos_svc = .;
}
```

Figure 2: *The linker script* `image.ld`*.*

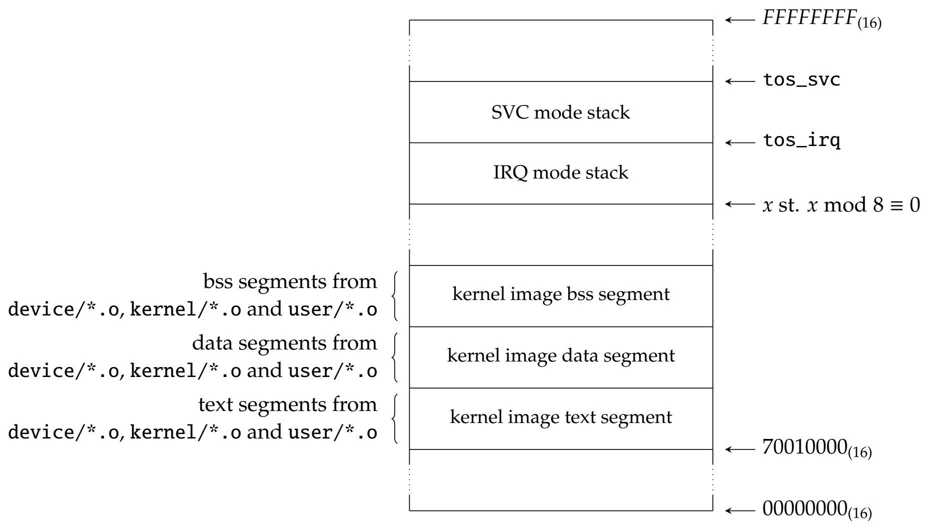

FFFFFFFF(16)
tos_svc
SVC mode stack
tos_irq
IRQ mode stack
x st. x mod 8 ≡ 0
bss segments from device/*.o, kernel/*.o and user/*.o
kernel image bss segment
data segments from device/*.o, kernel/*.o and user/*.o
kernel image data segment
text segments from device/*.o, kernel/*.o and user/*.o
kernel image text segment
70010000(16)
00000000(16)

Figure 3: *A diagrammatic description of the memory layout realised by* `image.ld`*.*

`int.[sh]`: low-level support functionality The header file `int.h` and source code `int.s` are essentially identical to worksheet #4 (bar the specialisation to interrupt types handled), so we omit any discussion of them.

`lolevel.[sh]`: low-level kernel functionality The header file `lolevel.h` and source code `lolevel.s` are essentially identical to worksheet #4 (bar the specialisation to interrupt types handled), so we omit any discussion of them.

`hilevel.[ch]`: high-level kernel functionality All high-level, kernel-specific functionality is captured by the header file `hilevel.h` and source code `hilevel.c`. Note that Line #10 of the latter matches the introduction above, allocating a 480000-element array of 16-bit integers to act as the frame buffer. The rest of the source code implements two high-level interrupt handler functions:

- `hilevel_handler_rst` is invoked by `lolevel_handler_rst` every time a reset interrupt is raised and needs to be handled:
  - Lines #15 to #25 a) configure the LCD controller to use the 800×600 resolution required, b) copy the frame pointer address into place, c) select a TFT-based display, opting for a 16BPP and hence $5:5:5$ pixel format, before d) enabling the controller and hence display.
  - Lines #36 to #53 configure the interrupt mechanism. This task is arguably more involved, although similar to previous worksheets so we look at it step-by-step:
    - Lines #36 to #39 configure the emulated PS/2 controllers, namely the `PL011_t` instances `PS0` and `PS1`, so each one raises an interrupt when they receive a byte from the connected device (e.g., as the result of a key-press).
    - Lines #41 to #46 represent a very partial, simple example of the PS/2 protocol. They act to enable both connected devices (which assumes there *are* such connections) by transmitting the relevant PS/2 command and waiting for an acknowledgement.
    - Lines #48 to #51 configure the GIC.
    - Line #53 then enables IRQ interrupts wrt. the processor.
  - Finally, Lines #57 to #61 represent two nested loops: in combination, they iterate through and assign values to all 480000 elements in the frame buffer. Noting that

    $$\texttt{0x1F << ((i / 200) * 5)} = \begin{cases} 001F_{(16)} & \text{for } \quad 0 \leq \texttt{i} < 200 \\ 03E0_{(16)} & \text{for } 200 \leq \texttt{i} < 400 \\ 7C00_{(16)} & \text{for } 400 \leq \texttt{i} < 600 \end{cases}$$

    the goal is to display a simple, banded test pattern: we expect red, green and blue regions in the top, middle and bottom thirds of the display.

- `hilevel_handler_irq` is invoked by `lolevel_handler_irq` every time a IRQ interrupt is raised and needs to be handled. The core of the function is represented by Lines #73 to #90, which handle the specific interrupt types:
  - Lines #74 to #80 handle interrupts generated by PS/2 controller #0, i.e., that connected to the keyboard device.
  - Lines #83 to #89 handle interrupts generated by PS/2 controller #1, i.e., that connected to the mouse device.

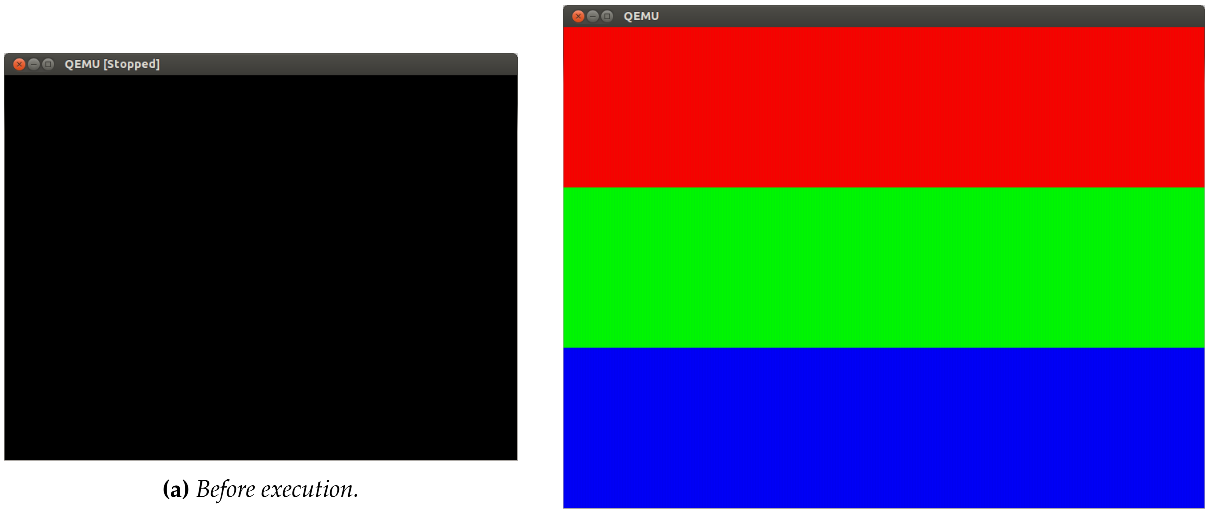
  Both cases perform essentially the same steps: they a) read the available byte from the controller (e.g., Line #74), then b) write some output which identifies the controller (e.g., Line #76) and includes the byte read from it (e.g., Lines #78 and #79).

Experiment with the archive content

Following the same approach as worksheet #1, first launch QEMU: you should find that a window matching Figure 4a appears. Note that closing the window will terminate QEMU. Next, launch `gdb` and issue the

```
continue
```

command in the debugging terminal so the kernel image is executed. Now, because `hilevel_handler_rst` is executed, the window is a) resizes to reflect configuration of the LCD controller, and, more obviously, b) matches Figure 4b due to the frame buffer being filled by the stated test pattern. Click on the window title



(a) *Before execution.* **(b)** *After execution.*

Figure 4: *The QEMU display before (left) and after (right) execution of* `hilevel_handler_rst` *populates the frame buffer. Note that the difference in window dimensions relates to configuration of the display, by* `hilevel_handler_rst`*, to use a resolution of* 800×600*.*

bar, st. it gains the UI focus. At this point any keyboard and mouse input to the window is communicated to QEMU, and thus, via appropriate interrupts, the kernel. For example, press the 'B' key: you should see

```
0<30>0<B0>
```

displayed, which means that the PS/2 controller connected to the keyboard generated two IRQ interrupts. The two bytes read, i.e., $30_{(16)}$ and $B0_{(16)}$ can be decoded as meaning "the 'B' key was pressed" and "the 'B' key was released".

Next steps

There are various things you could (optionally) do next: here are some ideas.

a The obvious next step is to parse the bytes received, essentially implementing the PS/2 protocol. For example you could parse the keyboard communication and render key-presses on the display using a bit-mapped[6] font, or parse the mouse communication and render a mouse cursor on the display which tracks movement.

b Rendering single pixels or bit-mapped shapes is reasonable simple, since you can pre-compute the data required and copy it into the frame buffer. A more generic approach would be to develop a small graphics library, e.g., to draw outlined or filled shapes (such as lines and circles). This is potentially more challenging than you would expect, but, equally, standard algorithms [3] exist that can help.

References

[1] ARM Limited. *PrimeCell Color LCD Controller (PL111) Technical Reference Manual*. Tech. rep. DDI-0293C. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0293c/index.html`. 2005 (see pp. 1, 2).

[2] ARM Limited. *PrimeCell PS2 Keyboard/Mouse Interface (PL050) Technical Reference Manual*. Tech. rep. DDI-0143C. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0143c/index.html`. 2005 (see p. 1).

[3] J.E. Bresenham. "Algorithm for computer control of a digital plotter". In: *IBM Systems Journal* 4.1 (1965), pp. 25–30 (see p. 5).

[6] `http://wiki.osdev.org/VGA_Fonts`